ACTIVE VERIFICATION OF BOOT FIRMWARE

[0001] This application claims priority from U.S. Provisional Application Serial No. 60/479,657 and 60/479,809 filed June 18, 2003, the contents being incorporated herein by reference.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] This invention was made with Government support under Contract F30602-02-C-0161 awarded by the Air Force and Contract DAAH01-02-C-R080 awarded by the U.S. Army Aviation and Missile Command. The Government has certain rights in this invention.

TECHNICAL FIELD

[0003] The invention relates to computer systems and, more particularly, to techniques for improving resistance to attacks on computer systems.

BACKGROUND

[0004] During startup, a typical computing device, such as a desktop computer, invokes a boot process to initialize the computing device and associated peripheral devices. The process of initializing each peripheral device is typically controlled by software, referred to as boot firmware, which is often stored in a read-only memory (ROM) on the computing device or on the respective peripheral device. After the computing device executes the boot firmware associated with each peripheral device, the computing device launches an operating system and possibly other software applications.

[0005] Until the device loads the operating system and any subsequent security software applications that rely on the operating system, the computing device may be susceptible to attack. In particular, it is possible for boot firmware to introduce serious security threats either accidentally or maliciously prior to the execution of the operating system and any security software applications that otherwise may neutralize the posed security threat. As a result, by the time the operating system and security software applications are available, the

security of the computing device may have already been compromised. Most conventional computing devices offer little resistance to security threats introduced by boot firmware.

[0006] In response to potential boot firmware security breaches, some computing devices provide security measures to ensure that the boot firmware comes from a trusted source. These security measures rely on digital signatures, which uniquely identify the source of the associated boot firmware. The computing device can decode a digital signature to identify the source of the boot firmware and accept or reject the boot firmware based on the source. In this manner, the computing device can gauge the reliability of boot firmware based on the source, allowing the computing device to execute only boot firmware from trusted sources.

[0007] One potential deficiency of this approach is that these approaches do not scale well. Typically, there are numerous sources generating boot firmware and the number of sources is growing. Furthermore, every new source must be verified so that the associated boot firmware may be executed. Thus, this approach yields to difficulties that arise as the number of sources grows, since the approach is dependent on the source.

[0008] Another potential deficiency of this approach is that the computing device may verify the reliability of the boot firmware only once. More specifically, the computing device receives the boot firmware and associated digital signature from the peripheral device, decodes the digital signature to accept or reject the boot firmware and executes only the accepted boot firmware. Thereafter, however, the computing device executes the boot firmware and assumes that the boot firmware is from the reliable source and has not changed. Other security measures may determine changes since the boot firmware was accepted and executes only boot firmware that has not changed from the accepted boot firmware. A computing device may use these two security measures in conjunction in an attempt to prevent the execution of boot firmware from unreliable sources, which may be malicious in nature, and prevent reliable boot firmware from being altered with malicious intent.

[0009] However, these security measures are passive measures, enforcing security by preserving trust, such as verifying the reputation of a source. Boot firmware from a malicious source can be installed under the guise of a reliable source, e.g., by misappropriating the digital credentials of the reliable source. In addition, even reliable sources can produce boot firmware that poses security threats accidentally through

programmer error. Both of these pose threats which passive security measures are incapable of preventing.

SUMMARY

[0010] In general, the invention is directed to techniques for ensuring safe operation of boot code in a computer system. The techniques describe processes to generate and verify boot code such that the computer system may only execute boot code that meets specified safety standards. The techniques may be useful in determining the safety of boot code independent of the reliability of a source that generated the boot code, thereby establishing trust.

[0011] In accordance with the principles of the invention, a certifying compiler can be used to generate a boot code and a certificate that allows a verification module to quickly verify the safe operation of the boot code. The certifying compiler may further comprise a program to generate the boot code and the certificate. Upon completion of the boot code and certificate, both may be loaded into a memory module of a peripheral device for use during initialization.

[0012] During initialization, the peripheral device having the memory module that stores the boot code and the associated certificate, communicates the boot code and the associated certificate to a computer system. The computer system executes the verification module, which actively verifies the security and safe operation of the boot code with aid from the certificate. The verification module may perform a security check to actively ensure a variety of specified safety standards are met. Upon completion of the verification process, the verification module either declares the boot code safe or unsafe, dependant on the outcome of the security check. In this manner, the certifying compiler and verification module enable the computer system to actively gauge the safety of the boot code independent of the source. Furthermore, since the verification module may verify the safety of the boot code independent of the source, issues of scale may not affect the verification module.

[0013] In one embodiment, the invention is directed to a method comprising verifying security of a boot code associated with a peripheral device by performing a security check on the boot code in accordance with a certificate that describes operation of the boot code and executing the boot code based on a result of the security check.

[0014] In another embodiment, the invention is directed to a method comprising generating a boot code for a peripheral device from a program written in a high-level programming language and generating a certificate from information gathered while generating the boot code, wherein the certificate describes operation of the boot code.

[0015] In yet another embodiment, the invention is directed to a device comprising a control unit to verify security of a boot code associated with a peripheral device by performing a security check on the boot code in accordance with a certificate that describes operation of the boot code and a memory module whereby the control unit executes the boot code based on a result of the security check.

[0016] In a further embodiment, the invention is directed to a device comprising a control unit to generate a boot code for a peripheral device from a program written in a high-level programming language and generate a certificate from information gathered while generating the boot code, wherein the certificate describes operation of the boot code.

[0017] In yet another embodiment, the invention is directed to a system comprising a peripheral device having a first memory module, wherein the first memory module stores a boot code and a certificate and a computer having a second memory module and a control unit, which retrieves the boot code and the certificate associated with the peripheral device and executes a verification module, wherein the verification module verifies security of the boot code by performing a security check of the boot code in accordance with a certificate that describes operation of the boot code and the control unit further executes the boot code based on a result of the security check.

[0018] In yet another embodiment, the invention is directed to a system comprising a peripheral device having a memory module, and a control unit to generate a boot code from a program written in a high-level programming language. The control unit also generates a certificate from information gathered while generating the boot code, wherein the certificate describes operation of the boot code. The control unit further loads the boot code and the certificate into the memory module.

[0019] In yet another embodiment, the invention is directed to a computer-readable medium containing instructions. The instructions cause a programmable processor to verify security of a boot code associated with a peripheral device by performing a security check on the boot

code in accordance with a certificate that describes operation of the boot code and execute the boot code based on a result of the security check.

[0020] In yet another embodiment, the invention is directed to a computer-readable medium containing instructions. The instructions cause a programmable processor to generate a boot code for a peripheral device from a program written in a high-level programming language and generate a certificate from information gathered while generating the boot code, wherein the certificate describes operation of the boot code.

[0021] In yet another embodiment, the invention is directed to a method comprising generating a boot code in the fcode programming language for a peripheral device from a program written in the Java programming language and generating a certificate from information gathered while generating the boot code, wherein the certificate describes operation of the boot code.

[0022] The details of one or more embodiments of the invention are set forth in the accompanying drawings and the description below. Other features, objects, and advantages of the invention will be apparent from the description and drawings and from the claims.

BRIEF DESCRIPTION OF DRAWINGS

[0023] FIG. 1 is a block diagram illustrating a computer system in which a central processing unit is connected to a plurality of peripheral devices.

[0024] FIG. 2 is a block diagram illustrating an exemplary computer system comprising a computer connected to a single peripheral device.

[0025] FIG. 3 is a flowchart illustrating an exemplary process whereby a computing device actively verifies the security of device drivers.

[0026] FIG. 4 is a flowchart illustrating an exemplary process whereby a verification module actively ensures the safety of device drivers by applying security checks in a three-tier process.

[0027] FIG. 5 is a block diagram illustrating an example computer system, wherein a source computer generates a device driver for a peripheral device.

[0028] FIG. 6 is a flow chart illustrating an exemplary process whereby a source code for a device driver written in a high-level language is compiled into a low-level language device driver.

DETAILED DESCRIPTION

[0029] For purposes of illustration, this disclosure refers extensively to methods for ensuring safe operation of boot firmware device drivers within a computer. In some embodiments, however, the invention may be applicable to ensuring safe operation of boot firmware modules, which includes but is not limited to device drivers. Accordingly, a description of verification and generation of device drivers to ensure safe operation within this disclosure should not be considered limiting of the invention as broadly claimed and embodied herein.

[0030] FIG. 1 is a block diagram illustrating a computer system 10 in which a central processing unit CPU 12 is connected to a plurality of peripheral devices 14A-14N (collectively "peripheral devices 14"). CPU 12 is connected to peripheral devices 14 via a communication interface such that CPU 12 may receive device drivers (not shown) stored within peripheral devices 14. CPU 12 can further operate in accordance with the Open Firmware standard as defined by IEEE-1275, which specifies a process for retrieving the device drivers from peripheral devices 14. Other standards may specify processes for retrieving the device drivers and CPU 12 is not limited to the Open Firmware standard. Moreover, the techniques may be applied to device drivers associated with peripheral devices 14 but centrally stored in a computer-readable medium, such as a boot disk.

[0031] In general, device drivers may be viewed as programs that specify a layer of abstraction to peripheral devices 14 so that higher-level software can access peripheral devices 14 in a uniform fashion. In particular, each device driver specifies an application program interface (API) to provide a mechanism for the higher-level software of CPU 12 to access the particular peripheral device. CPU 12 retrieves the device drivers from peripheral devices 14 and, in accordance with the techniques described herein, verifies that the device drivers correctly follow procedures for safe operation according to specified standards for the particular peripheral device and in doing so verifies that the device drivers properly define an API. After verification of the device drivers is complete, CPU 12 executes the device drivers based on the outcome of the verification.

[0032] Peripheral devices 14 may include a wide range of devices, such as graphic devices, network controllers and storage controllers, all of which contain different device drivers to allow full functionality of the varying peripheral devices. Exemplary network controllers

may include 10 megabit Ethernet controllers, 10/100 megabit Ethernet controllers, Infiniband controllers, iSCSI ("Internet Small Computer System Interface") controllers and the like. Exemplary storage controllers may include IDE/ATA controllers, Serial ATA controllers, SCSI controllers ("Small Computer System Interface"), Fibre Channel controllers, and the like. The verification process, carried out by a program referred to as a verification module, may distinguish device drivers for different devices and analyze the device driver based on the particular peripheral device. Thus, the verification module is a comprehensive program that receives a device driver and analyzes the device driver based on specified criteria corresponding to the particular peripheral device, as described in a certificate associated with the device driver.

[0033] The verification techniques are described in more detail in U.S. Provisional Application # 60/479,657, entitled "Compilation and Verification of Boot Firmware", filed June 18, 2003, the entire content of which are hereby incorporated by reference.

[0034] Since the verification module compares the device driver against the certificate, security is actively ensured independent of a source that generated the device driver. Thus, the verification module actively gauges the reliability of a source that generated the device driver. Moreover, issues of scale do not arise since the verification module is independent of the source. In instances where highly reliable sources unintentionally generate device drivers that threaten security, the verification module can determine the device driver as a security threat with aid from the corresponding certificate, and prevent CPU 12 from executing the driver despite the high reliability of the source. Furthermore, the verification module may detect malicious code despite the source, thus preventing the installation of malicious code under the guise of a reliable source.

[0035] The verification module may further perform the verification process on each device driver every time CPU 12 executes a device driver. The process of executing boot device drivers is referred to as the boot process, which occurs every time CPU 12 is reset or powered-up. CPU 12 may execute the verification module during the boot process such that no device driver is executed without first having the verification module verify the device driver. Also, CPU 12 can execute the verification module to analyze device drivers not executed during the boot process, e.g., a plug-and-play method, such that no device driver is

executed without first having the verification module examine the device driver. Thus, the verification module can ensure safety each time CPU 12 executes a device driver.

[0036] Although described for exemplary purposes herein in reference to a firmware-based module, the verification process may be implemented in software, firmware, hardware, or combinations thereof.

[0037] FIG. 2 is a block diagram illustrating an exemplary computer system 20 comprising a computer 22 connected to a single peripheral device 24. Computer 22 is connected to peripheral device 24 via a communication interface 25, such as an input/output (I/O) bus, for accessing memory module 28 of peripheral device 24. In particular, CPU 26 retrieves a device driver 32 from memory module 28 and executes verification module 30, which analyzes device driver 32 and determines whether device driver 32 is safe to execute.

[0038] Device driver 32 comprises boot code 36, which specifies an API for accessing and controlling peripheral device 24. In addition, device driver 32 includes a certificate 38 to aid verification module 30 in determining whether operation of boot code 36 is safe. Memory module 28 may comprise read-only memory (ROM) such as Programmable ROM (PROM), Erasable Programmable ROM (EPROM), Electrically Erasable Programmable ROM (EEPROM), and the like.

[0039] Upon receiving device driver 32 from peripheral device 24, verification module 30 proceeds to verify boot code 36 using certificate 38. For example, verification module 30 may perform a three-tier security process to verify boot code 36. Each tier is dependent on the prior tiers, and comprises a series of security checks. Should a security check within a tier fail, verification module 30 declares boot code 36 as unsafe and CPU 26 does not execute boot code 36. However, if verification module 30 declares boot code 36 as safe, i.e., boot code 36 passes the security checks of each tier, CPU 26 executes a copy of boot code 36. Boot code 40 represents a copy of boot code 36 and CPU 26 loads a copy of boot code 36 into memory module 34 as boot code 40. Consequently, boot code 40 comprises a series of instructions to initialize peripheral device 24 and upon execution of these instruction by CPU 26, higher-level programs, e.g., an operating system executing on CPU 26, interact with peripheral device 24. In this manner, boot code 40 enables and provides for interactions between computer 22 and peripheral device 24.

[0040] In some embodiments, computer 22 may connect to a plurality of peripheral devices, as illustrated in FIG. 1. Furthermore, verification module 30 may perform the three-tier security process on a plurality of device drivers associated with the plurality of peripheral devices. The process proceeds as described above, wherein upon receiving device drivers from each peripheral device, verification module 30 performs the three-tier security process, declares each boot code associated with a device driver as safe or unsafe, and CPU 26 executes a copy of the boot code. Moreover, CPU 26 may execute verification module 30 at any time for "inline" verification. For example, CPU 26 may execute verification module 30 during a boot process or upon executing device drivers during normal operation, e.g., plug-and-play, to verify the safety of the device driver before executing the boot code. In this manner, verification module 30 actively verifies security for each device driver before executing the associated boot code, independent of a source that generated the device driver.

[0041] FIG. 3 is a flow chart illustrating an exemplary process whereby a computing device, e.g., computer 22 of FIG. 2, actively verifies the security of device drivers. Specifically, computer 22 performs the illustrated exemplary process to verify device driver 32 stored on memory module 28 of peripheral device 24.

[0042] Computer 22 begins the process by retrieving device driver 32 (50). CPU 26 may execute instructions to establish communication with peripheral device 24 to retrieve device driver 32. CPU 26 may further conform to standards, such as the Open Firmware standard as define by IEEE-1275, which specifies a protocol for retrieving device driver 32. Upon retrieving device driver 32, CPU 26 executes verification module 30 (52).

[0043] Verification module 30 performs a series of security checks (54), which may be implemented in three tiers, on device driver 32. As described above, device driver 32 comprises boot code 36 and certificate 38. Each tier is dependant on the prior tiers and comprises a series of security checks. Verification module 30 applies each tier to boot code 36 with aid from certificate 38 to determine whether a copy of boot code 36 is safe to execute (56). Certificate 38 aids verification module 30 by indicating where in boot code 36 to apply the security checks. Certificate 38 may also specify the type of security check to perform at the specified location within boot code 36. If boot code 36 fails any of the security check then boot code 36 is declared unsafe and is not executed thereby preventing security threats. However, in the event that boot code 36 passes all of the security checks, verification module

30 declares boot code 36 as safe and CPU 26 executes a copy of boot code 36, i.e., boot code 40.

[0044] FIG. 4 is a flow chart illustrating an exemplary process whereby verification module 30 actively ensures the safety of device drivers by applying security checks in a three-tier process. Each tier of the three-tier process comprises a series of security checks to ensure the safety of device drivers. In this manner, computer 22 (FIG. 2) may rely solely on verification module 30 to provide active security verification of device drivers.

[0045] Specifically, verification module 30 applies the three-tier process to device driver 32, which CPU 26 retrieves from peripheral device 24 for processing by verification module 30. Each tier is applied to boot code 36 with aid from certificate 38, both of which are associated with device driver 32. Furthermore, each tier is dependent on the prior tiers, except the first tier, which comprises basic safety security checks. Should any security check included within a tier fail, verification module 30 need not apply any further security checks and declares boot code 36 unsafe. However, if boot code 36 passes each tier of security checks then verification module 30 declares boot code 36 safe.

[0046] The description as follows uses the abstraction of tiers to represent application of security checks by verification module 30 in a set of stages or phases. The tier abstraction aids in defining a mechanism for applying checks and aids in illustrating principles of the inventions, but other techniques for application of security checks within computer 22 are readily applicable. For example, verification module 30 may apply security checks without any recognition of tiers. Instead, verification module 30 may seamlessly apply security checks in a manner to resolve tier dependency, one after another until either boot code 36 passes all security checks or fails a security check.

[0047] Verification module 30 applies security checks according to certificate 38, which indicates lines of code within boot code 36 to begin applying the security checks. Certificate 38 may further define the type of security check for verification module 30 to apply. Consequently, using certificate 38, verification module 30 may quickly perform security checks to pertinent portions of boot code 36 without having to analyze every portion of boot code 36. Since certificate 38 allows verification module 30 to perform security checks without analyzing every portion of boot code 36, the computing time required to verify boot code 36 is reduced and, consequently, CPU 26 may execute verification module 30 "inline"

with the execution of device drivers. "Inline" active verification of device drivers may provide assurance that no device driver is executed without first being verified.

[0048] CPU 26 executes verification module 30 after retrieving device driver 32 from memory module 28. Verification module 30 receives device driver 32, which comprises boot code 36 and certificate 38 (60) from CPU 26. Upon receiving device driver 32, verification module 30 begins applying tier one security checks (62) according to certificate 38. The first tier uses Efficient Code Certification (ECC) providing a basic security policy to ensure type-safety. ECC is a form of language-based security employing inexpensive static checks on boot code and certificates in order to verify dynamic safety properties. Tier one may apply safety checks very similar to that of a Java verifier program which also ensures type-safety. Specifically tier one may ensure type safety, control flow safety, memory safety and stack safety, similar to the verification of Class files performed by the Java Virtual Machine.

[0049] Certificate 38 comprises information collected during compilation of boot code 36, e.g., by a compiler, as described in further detail below. Verification module 30 applies the static checks using principles of ECC to boot code 36 according to certificate 38, thereby ensuring type safety, control flow safety, memory safety and stack safety. Specifically, verification module 30 ensures control flow safety using static security checks to verify that boot code 36 only accesses addresses containing valid instructions within boot code 36. Verification module 30 also ensures memory safety by, again, applying static checks to verify that boot code 36 only causes CPU 26 to accesses valid locations within the data segment CPU 26 assigned to boot code 36, system heap memory CPU 26 explicitly allocated to boot code 36, and valid stack frames. Verification module 30 applies further static checks, which verify that boot code 36 properly preserves the stack across all subroutine calls, to ensure stack safety.

[0050] Verification module 30 applies the tier one security checks to boot code 36 with aid from certificate 38 based on principles of ECC and evaluates the result of each individual security check (64). If boot code 36 fails to pass any of the security checks of tier one, then verification module 30 declares boot code 36 unsafe (74) and does not continue applying any further security checks. No further security checks are applied since each tier is dependant on the prior tiers, thus failing tier one implies failure of tier two and tier three. However,

verification module 30 continues to apply tier two security checks if boot code 36 passes all the security checks associated with tier one (66).

[0051] ECC may further aid in both tier two and tier three security checks. Verification module 30 using ECC may perform lightweight, inexpensive security checks, which are included within tier two and three. Since the security checks are relatively inexpensive, CPU 26 may execute verification module 30 prior to executing device drivers, either before a boot process or before executing device drivers during normal operation, such as is done by the plug-and-play method. Again, this "inline" execution of verification module 30 may ensure that no device drivers are executed without first being verified.

[0052] Tier two security checks comprise security checks to ensure device encapsulation. Ensuring device encapsulation involves a process verification module 30 performs to ensure that each peripheral device, such as peripheral device 24, is operated directly or indirectly only by the device driver associated with the peripheral device, such as device driver 32 associated with peripheral device 24. For the verification of directly operated devices, verification module 30 ensures that only the device driver associated with a peripheral device actually is used to operate the device. Some devices, such as those connected to the processor via expansion busses, are only operated indirectly via a chain of other devices, wherein the device drivers associated with these devices access the bus. These prove harder for verification module 30 to verify. However, indirect access methods are pre-specified and highly controlled, providing verification module 30 with a basis to verify boot code of such indirectly operated devices.

[0053] Verification module 30 may verify device encapsulation for indirectly operated devices by observing a pattern of calls to a program procedure, such as the *mapin* procedure specified by the Open Firmware standard, which perform address translation in a standardized manner. Verification module 30 may analyze the pattern of program procedure calls and compare them with the highly controlled and pre-specified methods for calling this program to determine appropriate use of the peripheral device, such as peripheral device 24. Furthermore, verification module 30 can verify that the above address translation procedure only accesses addresses allocated to the particular device. Verification module 30 may accomplish the above quickly using certificate 38, which ensures that all indirect accesses occur using a pre-specified verification API, which is described below. Using this

verification API ensures proper security, since the verification API follows the specified and highly controlled standards meant to ensure safe indirect accesses.

[0054] After applying tier two security checks to ensure device encapsulation, verification module 30 may analyze the results of each security check. Again, verification module 30 declares boot code 36 as unsafe (74) once verification module 30 determines that boot code 36 fails any one of the security checks of tier two.

[0055] When boot code 36 passes all the security checks of tier two, verification module 30 applies the next tier of security checks, i.e., tier three (70). Tier three security checks allow verification module 30 to protect against specific harm. Tier three security checks may be based on architectural constraints or dependent upon standards, such as the Open Firmware standard as defined by IEEE-1275. For example, tier three security checks may comprise security checks to ensure device driver 32 does not access a device more than a pre-determined number of times. By restricting device accesses to the pre-determined number, verification module 30 may prevent denial-of-service attacks. Tier three security checks may further prevent specific security threats other than denial-of-service attacks.

[0056] After applying tier three security checks, verification module 30 determines whether boot code 36 is safe or unsafe (72). Verification module 30 examines the results and declares boot code 36 unsafe (74) if boot code 36 fails a single tier three security check. Otherwise, verification module 30 declares boot code 36 safe (76).

[0057] Each of the tiers as described above comprises security checks enabling verification module 30 to declare boot code 36 as either safe or unsafe. If at any point boot code 36 fails a single security check then verification module 30 declares boot code 36 unsafe. Thus, verification module 30 is a comprehensive security verification program that actively checks the safety of boot code 36 using certificate 38 to quickly verify whether boot code 36 is safe. Furthermore, CPU 26 may execute verification module 30, which utilizes certificate 38 as described above, inline with the device driver execution, thus ensuring device driver safety each time CPU 26 executes a device driver. Verification module 30 may use a verification process comprising three tiers as described above or may verify boot codes using any other process consistent with the principles of the invention.

[0058] FIG. 5 is a block diagram illustrating an example computer system 80, wherein a source computer 82 generates a device driver 108 for a peripheral device 84. Specifically,

source computer 82 generates device driver 108 via a process whereby a series of programs process high-level program 90 to generate certificate 99 and associated boot code 106. Device driver 108, comprising certificate 99 and boot code 106, may then be loaded onto memory module 88 of peripheral device 84.

[0059] The compilation process from a high-level language program, e.g., high-level program 90, to boot code 106 may comprise any number of steps and involve a plurality of programs in cooperation to achieve the compilation process. The process described herein is an example of one embodiment of the invention and should not be considered as a sole representative of the invention on the whole. Furthermore, high-level program 90 may comprise object oriented high-level languages such as Java, C++, Visual Basic and the like. Also, boot code 106 may conform to a standard, such as the Open Firmware standard as defined by IEEE-1275, which may specify a particular code language, such as fcode, and format.

[0060] The compilation techniques are described in more detail in U.S. Provisional Application # 60/479,657, entitled "Compilation and Verification of Boot Firmware", filed June 18, 2003, the entire content of which is hereby incorporated by reference.

[0061] Source computer 82 comprises CPU 84 and memory module 86. CPU 84 may execute various programs which accesses memory module 86 and may further write data to addresses within memory module 86. The process of compiling high-level program 90 into certificate 99 and boot code 106 may begin once high-level program 90 is complete and an operator of source computer 82 instructs CPU 84 to begin the process.

[0062] CPU 84 begins the compilation process of high-level program 90 by executing high-level compiler 92 and accessing memory module 86 in order to retrieve high-level program 90. High-level compiler 92 compiles code specific to the high-level language used to construct high-level program 90. For example, if high-level program 90 is written in Java, then high-level compiler 92 performs some of the functions of typical Java compilers. Furthermore, high-level compiler 92 has knowledge of verification API 94 and specifies compilation instructions pertinent to verification API 94.

[0063] Prior to compiling high-level program 90, compiler 92 analyzes high-level program 90 and verifies that high-level program 90 adheres to a pre-determined safety policy. Specifically, high-level program 90 corresponds to a device driver written in a high-level

language and high-level compiler 92 verifies that the device driver conforms to all or part of the three-tier security policy discussed above. During compilation of high-level program 90, high-level compiler 92 processes instructions associated with high-level program 90 to construct bytecode 96 and may verify proper use of verification API 94 encoding relevant information to certificate 98. High-level compiler 92 may construct certificate 98 to include information gathered during the compilation of bytecode 96. In some embodiments certificate 98 is produced as a piece of data separate from bytecode 96. Other embodiments may produce certificate 98 as an integral part of bytecode 96. In these latter embodiments, the information of certificate 98 is incorporated in such a way that it is easily extractable from the combination.

[0064] High-level compiler 92 with knowledge of verification API 94 also generates certificate 98. Certificate 98 is an annotation of blocks of bytecode 96, which constitutes proof of why bytecode 96 and eventually boot code 106 meet specified safety conditions. For example, high-level compiler 92 may determine type information corresponding to a variable in high-level program 90 and incorporate this information into certificate 98. The annotations of certificate 98 assert proof, which a verification module, such as verification module 30 of FIG. 2 may verify, that a block of bytecode 96 and a resulting block of boot code 106 are safe. To continue the above example, verification module 30 may quickly check the type of the variable using the type information corresponding to the variable found in certificate 98. Thus, verification module 30 may quickly apply the security check since verification module 30 does not need to re-determine the type of the variable. In full, verification module 30 may quickly verify boot code 106 such that if all the assertions prove true then verification module 30 may declare boot code 106 as safe.

[0065] In some embodiments, boot code 106 may conform to a standard, such as the Open Firmware standard. Some standards specify a specific code language, such as fcode, for which boot code 106 conforms to follow the standard. Bytecode 96 typically is not a suitable boot code and CPU 84 may further process bytecode 96 to generate boot code 106 such that boot code 106 conforms to a standard.

[0066] In particular, most device drivers and boot firmware are written in low-level languages, such as Forth. Translator 100 interprets bytecode 96 and generates low-level code 102. Furthermore, most low-level languages are not object-oriented languages, thus,

translator 100 must further process object-oriented bytecode 96 to generate non-object oriented low-level code 102.

[0067] During translation, translator 100 may analyze certificate 98, together with bytecode 96, gather additional information necessary to complete the three tiered verification process described above and add this information to certificate 98. Again, this information may allow a verification module, such as verification module 30, to quickly ascertain the safety of boot code, as described above. Translator 100 then writes certificate 98 with the new additions, possibly translated to a new format, to memory module 86, shown in FIG. 5 as certificate 99. In some embodiments certificate 98 is produced as a piece of data separate from bytecode 96. Other embodiments may produce certificate 98 as an integral part of bytecode 96. In these latter embodiments, the information of certificate 98 is incorporated in such a way that it is easily extractable from the combination.

[0068] An additional program, i.e. tokenizer 104, may also process low-level code 102 to generate boot code 106. CPU 84 may again execute tokenizer 104 to process low-level code 102 to generate boot code 106, such that boot code 106 conforms to a standard. Tokenizer 104 analyzes low-level code 102 and generates boot code 106 in a format, e.g. a ROM image, that allows source computer 82 to install boot code 106 onto memory module 88.

[0069] Upon completion of the compilation process, i.e., completion of the execution of tokenizer 104, source computer 82 may install boot code 106 and certificate 99 to memory module 88 of peripheral device 84. Source computer 82 may "flash" certificate 99 and boot code 106 onto any of the above mentioned ROM types, which memory module 88 may represent, forming device driver 108. Memory module 88 is not limited to storing one device driver, i.e., device driver 108, but may store many device drivers corresponding to various device aspects associated with peripheral device 84. Furthermore, memory module 84 may store any other data, including data from computers other than source computer 82.

[0070] The process of writing high-level program 90, a device driver, in a high-level language, compiling the code with high-level compiler 92, and translating the code via translator 100 to low-level code 102 may greatly aid programmers who author device drivers. In particular, object oriented code provides for easy organization and allows programmers to work in groups to create a single program, such as high-level program 90. Low-level code 102, in most cases, does not suit group programming or convenient organization, thus the

above process may decrease the time needed to create device drivers by offering the ability to increase throughput through group work and organization.

[0071] FIG. 6 is a flow chart illustrating an exemplary process whereby a source code for a device driver written in a high-level language is compiled into a low-level language device driver. As one example, the device driver source code can be written in the Java high-level language and compiled into a low-level language, e.g., fcode, device driver through a process in which several computer programs cooperate to complete the compilation process. Consequently, in one embodiment, the fcode device driver conforms to the Open Firmware standard as specified by IEEE-1275.

[0072] The process as outlined above may proceed with CPU 84 of source computer 82 (FIG. 5) receiving the java device driver source code (120) as represented by high-level program 90. CPU 84 then executes high-level compiler 92 to compile the device driver written in Java, i.e., high-level program 90. High-level compiler 92 with knowledge of verification API 94 compiles the java device driver to generate Java Virtual Machine (JVM) bytecode, i.e., bytecode 96 (122). High-level compiler 92 may ensure certain functions associated with verification API 94 are called appropriately and encode the use of these functions into certificate 98.

[0073] CPU 84 stores the JVM bytecode to memory and executes translator 100. Translator 100 takes JVM bytecode and generates a Forth program (124), which is represented as low-level code 102. As discussed generally above, translator 100 converts the object oriented nature of the JVM bytecode into a forth program, which does not support objects. Furthermore, translator 100 may fix problems, such as lazy class loading and initialization that normally occur in Java, during translation. Translator 100 may also make additions to certificate 98 and store certificate 99, which incorporates the new additions, to memory module 86.

[0074] The forth program represented as low-level code 102 is finally tokenized into fcode, or boot code 106 (126). The process of tokenizing with respect to the forth program comprises compiling the forth program into fcode. This process is described in detail in the Open Firmware standard.

[0075] A certifying compiler may comprise these three steps whereby verification module 30 of FIG. 2 may verify the resulting fcode device driver to determine whether the resulting

fcode device driver is safe for computer 22 to execute. The certifying compiler that performs the above process ensures safety by verifying the java device driver prior to proceeding with the compilation process. If the java device driver is declared unsafe by the certifying compiler, then source computer 82 will not proceed with compiling of the java device driver, i.e., high-level program 90. Thus, only verifiably safe high-level programs are compiled by source computer 82.

[0076] The compilation process as described above may incorporate several programs, i.e., high-level compiler 92, translator 100, tokenizer 104, working in cooperation to compile high-level program 90 into boot code 106. Furthermore, the compilation process may generate certificate 99, allowing a verification module, such as verification module 30, to quickly ensure safe operation of boot code 106 prior to executing boot code 106. Using both a certifying compiler, comprising for example, the before mentioned several programs, and a verification module, active security measures may ensure the safe operation of boot code, such that the reliability of the boot code is determined dependent on the boot code and independent of the source.

[0077] Various embodiments of the invention have been described. These and other embodiments are within the scope of the following claims.